HalOS

Andreas Jung,

Andreas Mayr,

Christian Brändle,

Mathias Giacomuzzi,

Markus Speckle,

Karl Zerlauth

Inhalt

Inhalt

Inhalt

Inhalt

1 Prozesse:

Ein Prozess besteht unter anderem aus
* Programmzähler
* Registern
* Variablen
Da die Hardware nur über einen physikalischen Programmzähler verfügt, wird beim Wechsel der Prozesse der Programmzähler des aktuell laufenden Prozesses gesichert. (S 56 The Minix Book, Third Edition)
(Dieser Text ist noch etwas dürftig. Wird noch erweitert, ist ja ein iterativer Prozess:)

1.1 Prozesserzeugung:

(S 58 - 59, The Minix Book, Third Edition) Prozesse werden durch Systemaufrufe erzeugt. Der Systemaufruf teilt dem Betriebssystem mit, dass ein neuer Prozess erzeugt werden und welches Programm im Prozess ablaufen soll. Als Systembefehl zum Erzeugen eines neuen Prozesses könnte der fork-Befehl verwendet werden. Dieser Befehl erzeugt eine exakte Kopie des aufrufenden Prozesses. Danach kann der execve-Befehl aufgerufen werden um ein neues Programm in den Speicher des neu erstellen Prozesses zu laden. Der Grund für dieses Zwei-Phasenmodell liegt darin, dass dem neuen Prozess erlaubt wird seine File Description nach dem fork und vor execve anzupassen. (In wie fern dies für unser Betriebssystem relevant ist muss noch abgeklärt werden.

1.2 Prozess Terminierung:

Ein Prozess kann durch folgende Ereignisse beendet werden:
* Normal exit * Error exit * Fatal error * Killed by another process Normale Beendigung wird vom Prozess durch den Aufruf des exit-Systemaufrufs angezeigt. Beendigung durch Fehler kann unter anderem durch ungültige Programmanweisungen, Referenzieren von ungültigem Speicher oder Division durch Null vorkommen. Ein Prozess kann einen andern Prozess durch den Aufruf des kill-Systemaufrufs beenden.

1.2.1 Prozess Hirachie:

Noch offen bwz. ob überhaupt sinnvoll.

1.2.2 Prozess Zustände:

(S61-62, The Minix Book, Third Edition) Jeder Prozess befindet sich in einem von drei möglichen Zuständen:
* Running (der Prozess verwendet gerade die CPU) * Ready (Ausführbar; der Prozess wurde vorübergehend angehalten) * Blocked (der Prozess kann nicht ausgeführt werden, bis ein externes Ereignis eintritt)

(Operation System Concepts, hat auf S. 97 noch ein erweitertes Modell).

1.3 Zustandsübergänge:

1.3.1 (1) Running ? Blocked

Dieser Zustandsübergang tritt auf, wenn ein Prozess nicht weiter ausgeführt werden kann, beispielweise wenn der Prozess auf Eingabezeichen warten muss. Es gibt zwei Möglichkeiten diesen Zustandsübergang zu realisieren. Der Prozess könnte den Systemaufruf block oder pause aufrufen um in den Blockiert-Zustand zu gelangen. Eine andere Möglichkeit wäre es, dass das Betriebssystem dies automatisch erkennt und den Prozess in den Blockiert-Zustand überführt.

1.3.2 (2) Running ? Ready

Dieser Zustandsübergang wird durch den Scheduler durchgeführt, wenn der aktuell laufende Prozess seine Zeitscheibe verbraucht und die CPU einem anderen Prozess zugeteilt wird.

1.3.3 (3) Ready ? Running

Dieser Zustandsübergang wird ebenfalls vom Scheduler angestoßen. Ein Prozess im Ready-Zustand wird dabei vom Scheduler zur Ausführung gebracht, wenn er auf Grund des Scheduling-Algorithmus wieder an der Reihe ist.

1.3.4 (4) Blocked ? Ready

Ein Prozess wird vom Blocked in den Ready-Zustand überführt, wenn das Ereignis auf das er gewartet hat eingetreten ist. Ist kein anderer Prozess in Ausführung, dann kann der Prozess direkt in den Running Zustand übergehen. Wird jedoch momentan noch ein anderer Prozess ausgeführt, dann muss auf die CPU Zuteilung gewartet werden

1.3.5 Prozessimplementierung

(S 62, The Minix Book, Third Edition) Jeder Prozess wird im Betriebssystem durch einen Prozess Control Block repräsentiert. Das Betriebssystem speichert diese PCB?s in einer Prozesstabelle. Ein PCB enthält unter anderem folgende Informationen: * Prozesszustand * Programmzähler * Stackpointer * Speicherallokierung * Status der geöffneten Dateien * Schedulinginformationen * ? Interrupt Descriptor Table noch offen, bwz. Unklar. (S64, The Minix book, Third Edition)

1.4 Interprocess Communication (IPC):

(S 68, The Minix Book, Third Edition) Bei der Interprozesskommunikation kommt es auf drei Fäll an: * Wie kann erreicht werden, dass sich Prozess bei kritischen Aktionen nicht stören? * Wie kann ein Prozess Informationen an einen anderen Prozess senden? * Wie können Prozesse die eine gegenseitige Producer/Consumer Abhängigkeit haben geeignet sequenzialisiert werden?

1.4.1 Mutual Exclusion /Wechselseitigerausschluss

(S 71, The Minix Book, Third Edition) In diesem Abschnitt werden Möglichkeiten angesprochen, wie Mutual Exclusion erreicht werden kann.

1.4.2 Interrupts ausschalten:

(S 71, The Minix Book, Third Edition) Der Prozess könnte alle Interrupts ausschalten, gerade nachdem er eine kritische Region betreten hat. Da die CPU auf Basis von Clock-Interrupts von Prozess zu Prozess weitergereicht wird, kann sich der Prozess so sicher sein nicht unterbrochen zu werden. Dieser Ansatz ist jedoch nicht tragbar, da hier der Userprozess die Möglichkeit hat Interrupts auszuschalten und möglicherweise nicht wieder einzuschalten. Dieser Umstand würde das ganze System korrupieren. Das Ausschalten von Interrupts kann jedoch innerhalb des Betriebssystemkernels nützlich sein, falls Datenstrukturen aktualisieren werden müssen und Inkonsistenzen vermieden werden müssen.

1.4.3 Lock Variables:

(S 71, The Minix Book, Third Edition) Eine Lockvariable enthält eine 0, wenn die kritische Region noch nicht betreten wurde. Der Prozess setzt die Lock Variable, danach auf 1 und betritt die kritische Region. Möchte ein zweiter Prozess die kritische Region betreten, erkennt aber, dass die Lock Variable auf 1 steht, so wird dieser in einen Wartemodus gehen. Dieser Ansatz leidet jedoch an einer Schwachstelle. Ist beispielweise die Lock Variable auf 0 und erste Prozess möchte die kritische Region betreten, so wird dieser die Lock Variable auf 1 setzen wollen. Wird der Prozess jedoch genau in diesem Augenblick unterbrochen und ein zweiter Prozess möchte ebenfalls in die kritische Region eintreten, so wird dieser zweite Prozess die Lock Variable mit dem Wert 0 vorfinden. Der zweite Prozess kann so ebenfalls in die kritische Region eintreten.

1.4.4 Strict Alternation:

(S 72, The Minix Book, Third Edition)

```
Process0
While (TRUE){
     while(turn  != 0);
     critical_region();
     Turn = 1;
     noncritical_region();
}

Process1
While (TRUE){
     while(turn  != 1);
     critical_region();
     Turn = 0;
     noncritical_region();
}
```

Die Variable turn hält fest welcher Prozess die kritische Region betreten darf. Zu Beginn ist diese Variable auf 0 gesetzt, damit ist Prozess0 am Zug. Der Prozess 1 findet die Variable turn auf 0 gesetzt vor und wartet in der

Schleife bis er an der Reihe ist (busy waiting). Eine Sperre die durch Busy Waiting erreicht wird nennt man spin lock. Wenn der Prozess0 die kritische Region verlässt, setzt er die turn Variable auf 1, um es Prozess1 zu ermöglichen die kritische Region zu betreten. Dieser Ansatz verhindert alle Race Condition. Es kann jedoch zu einer Situation kommen, in der Prozess0 nicht die kritische Region betreten kann, obwohl Prozess1 sich nicht in der kritischen Region befindet (genauers auf S. 73, The Minix Book, Third Edition). Außerdem müssen bei diesem Ansatz die Prozesse jeweils alternierend die kritische Region betreten.

1.4.5 Petersons?s Solution:

(S 73, The Minix Book, Third Edition)

Diese Lösung kombiniert die Idee der turns mit dem Konzept der Lock Variablen. Dieser Ansatz löst das Mutual Exclusion Problem ohne strikte Alternierung (Strict Alternation). Auch dieser Ansatz basiert auf Busy Waiting. (Genauere Informationen auf S. 74, The Minix Book, Third Edition zu finden)

1.4.6 TSL Instruction

(S 75, The Minix Book, Third Edition) Diese Variante benötigt Hardware Unterstützung. Dabei kommt eine sogenannte TSL (Test and Set Lock) Instruktion zur Anwendung. Abklären ob AVR32 solche Instruktion hat. Alle diese Varianten, leiden am Problem des Busy Waitings. Im nachfolgenden werden Ansätze bestrochen, die kein Busy Wating erfordern. Diese basieren auf den Systemaufrufen sleep und wakeup. Semaphores: (S. 201, Operating System Concepts, Sixth Edition) Eine Semaphore S ist eine Integer Variable auf die nur durch zwei Systemaufrufe wait und signal zugegriffen werden kann. Die wait-Operation sieht in Pseudocode folgendermaßen aus:

```
wait(S){
      while(S <= 0);
      S--;
}
```

1.4.7 Die signal-Operation sieht in Pseudocode etwa so aus:

```
Signal(S){
      S++;
}
```

Die Modifikation der Integervariable in wait und signal muss unteilbar(atomar) sein.

1.4.8 Implementierung:

(S. 203, Operatin Sytem Concepts, Sixth Edition) Eine Semaphore wird durch folgende C-Struktur definiert:

```
typedef struct
{
      Int value;
      Struct process *L;
}semaphore;
```

Jede Semaphore hat eine Integervariable und eine Liste von Prozessen. Wenn ein Prozess auf eine Semaphore warten muss, dann wird er in die Liste der wartenden Prozesse eingetragen. Ein Aufruf des signal Systemaufrufs, entfernt ein Prozess aus der Liste der wartenden Prozesse und weckt diesen Prozess auf. Die wait-Semaphore Operation kann damit wie folgt definiert werden:

```
void wait(semaphore S)
{
     S.value--;
     if(S.value < 0){
          Add this process to S.L;
          block();
}
}
```

1.4.9 Die signal-Semaphore Operation kann damit folgendermaßen definiert werden:

```
void signal(semaphore S)
{
     S.value++;
     if(S.value <= 0){
          Remove a process P from S.L;
          wakeup(P);
}
}
```

Die block Operation suspendiert den Prozess der sie aufruft. Die wakeup(P) Operation ermöglicht es einem blockierten Prozess wieder ausgeführt zu werden. Diese zwei Operationen werden als Systemaufrufe vom Betriebssystem zur Verfügung gestellt. Die Liste der wartenden Prozesse kann durch Zeiger auf PCB?S (Prozess Control Blocks) realisiert werden. Für die Entnahme von Prozessen aus der Liste ist z.B. eine FIFO-Strategie geeignet.

1.4.10 Mutexes:

S. 81, The Minix Book, Third Edition

1.4.11 Monitors:

Monitore sind Programmiersprachenkonstrukte und aus diesem Grund nicht relevant für unsere Betrachtungen. (S 82, The Minix Book, Third Edition)

1.4.12 Message Passing

(S 85, The Minix Book, Third Edition) Dieser IPC Ansatz verwendet Systemaufrufe wie send und receive. Die Schnittstellen dieser Systemaufrufe könnten beispielsweise so aussehen: send(destination, &message); receive(source, &message); Die erste Methode sendet eine Nachricht an einen Empfänger und die zweite wartet auf eine Nachricht. Sollte beim Warten keine Nachricht vorhanden sein, kann der Prozess blockiert bis eine Nachricht ankommt oder es wird ein Fehlercode zurückgegeben und der Prozess kann weiterlaufen. Beim Message Passing

müssen die beteiligen Prozesse eindeutig identifizierbar sein. Ein Problem bei diesem System stell die Authentizität des senden bzw. empfangenen Prozesses dar. Die Performance von Nachrichten-Kopieroperationen wird in der Regel auch langsamer sein als die Nutzung von Semaphoren.

1.5 Scheduling:

2 HALOS I/O ? SYSTEM

2.1 1) Anforderungen

Möglichst einfach portierbar auf eine anders Zielsystem Wenn möglich einheitliche Schnittstellen zu den I/O Geräten (PIO, Timer, MMC, PS/2). Direkter Zugriff auf die Hardware soll verhindert werden.

2.1.1 2) Design ? Entscheidung

1)Für jedes physikalische Gerät welches and den Prozessor angeschlossen werden soll muss eine Device Driver implementiert werden. Der Driver übernimmt die low level Kommunikation mit dem Gerät und implementiert die Standart funktionen wie (open, init, read, write, ioctl).

2)Zudem findet eine Unterteilung in Block-, Byte- und Bit-Devices statt. Damit können die Geräte in einer standardisierten einheitlichen weise angesprochen werden.

2.1.2 2) IO spezifische System Calls:

2.1 open_driver

Dieser Systemaufruf wird zum Initialisieren eines Gerätetreibers verwendet. Als wichtige Argumente werden folgende Parameter übergeben:

Gerätekennung Durch die Gerätekennung werden die unterschiedlichen physikalischen Gerätetypen unterschieden (Serial Device, Timer Device, ?).

Gerätenummer Die Gerätenummer kennzeichnet ein physikalisches Gerät (COM1, COM2, Timer2, ?) .

Als Rückgabe wird eine Struktur mit unteranderem folgenden Informationen:

Gerätebezeichnung UID des Geräts Funktionspointer auf Init, Read,Write und Close Methoden. Je nach Geräte zeigt der Funktionspointer für die genannten Funktionen (z.B.: Read u. Write) auf Bit, Byte oder Block Funktionen.

2.6 writeblock_driver

Dieser Systemaufruf wird verwendet um einem Block Device Daten zu senden. Neben den Datenpointer wird auch die Struktur übergeben welche beim open_driver für den jeweiligen Driver geliefert wurde. Die beiden nachfolgenden Methoden writebyte_driver und writebit_driver funktionieren gleich wobei wie die Methoden schon sagen ein Byte oder ein Bit als Datum geschrieben wird.

2.2 writebyte_driver

siehe wirteBlock_driver

2.4 writebit_driver

siehe writeblock_driver **2.7 readblock_driver**

Dieser Systemaufruf wird verwendet um von einem Block Device Daten zu lesen. Übergeben wird die Struktur welche beim open_driver für den jeweiligen Driver geliefert wurde sowie eine Pointer auf einen Speicherblock in den geschrieben werden soll.

2.3 readbyte_driver

Dieser Systemaufruf wird verwendet um von einem Character Device Daten zu lesen. Übergeben wird neben dem Datenbyte die Struktur welche beim open_driver für den jeweiligen Driver geliefert wurde. sowie eine Pointer auf einen Speicherblock in den geschrieben werden soll.

2.5 readbit_driver

Dieser Systemaufruf wird verwendet um von einem Character Device Daten zu lesen. Übergeben wird neben dem Datenbit die Struktur welche beim open_driver für den jeweiligen Driver geliefert wurde. sowie eine Pointer auf einen Speicherblock in den geschrieben werden soll.

2.5 close_driver

Dieser Systemaufruf wird verwendet um einem Device Driver zu schliessen. Übergeben wird die Struktur welche beim open_driver für den jeweiligen Driver geliefert wurde.

2.6 ioctl_driver

Dieser Systemaufruf wird für die Steuerung eines Gerätetreibers verwendet. (Beispielsweise bei einer Seriellen Schnittstelle die Baudrate verändern, usw.) Ein Übergabeparameter ist die Struktur, welche beim open_driver für den jeweiligen Driver geliefert wurde. Ein weiterer Parameter ist das Kommando welches im Driver ausgeführt werden soll. Als dritter Parameter wird ein Pointer auf eine Konfigurationsstruktur übergeben (Die Baudrate die eingestellt werden soll steht in dieser Struktur).

2.2 Möglicher Ablauf beim lesen des Character Devices (Serielle Schnittstelle)

(1)Ein Prozess führt einen blockierenden Systemcall read_byte() aus, als Parameter wird die Device Stuktur welche bei device_open() geliefert wurde übergeben. Es findet ein Kontexwechsel vom User Mode in den Kernel Mode statt via ASM-Befehl ?scall?. Es wird ein scall-Interrupt ausgelöst und in den scall handler gesprungen.
(2)Je nach Übergabeparameter wird entschieden um welchen Befehl es sich handelt. Der rufenden Prozess wird von der run queue entfernt und auf die wait queue des Gerätetreibers verschoben. Der Request wird schließlich and den Device Driver übergeben. Sobald das Transfer ende erreicht ist wird von dem entsprechenden Device ein Interrupt ausgelöst. In der Interruptserviceroutine wird der Kernel benachrichtigt, dass der Transfer beendet ist. Der Kernel übergibt die Daten an den rufen Process und verschiebt den Prozess von der wait queue des Gerätes in die process ready queue des Kernels.
(3)Durch die Verschiebung wird der Prozessstatus von blocked auf unblocked gesetzt. Sobald das Scheduling durchgeführt wird wird dieser Prozess weitergeführt sofern es sich um den höchstprioren Process handelt.

3 HALOS ? Kernel

3.1 1) Anforderungen

Scheduling Weiche Echtzeit Preemtives Scheduling Multiprocessing ?

Virtuelle Speicher Paging Auslagern von Pages auf SD-Karten ?

Anwendungen dynamisch von SD-Karte Process Loader Anwendung muss beendet werden könne ?

Treiberkonfigurierbar und fix geladen Es muss nichts nachladbar sein ?

Bibliothek für Syscalls I/O Sonstige Aufgaben ?

Performance Counter einsetzten um Metrik für BS zu definieren um BS zu evaluieren ?

Kernel SW Architektur Modular Robust Einfach

3.2 Design ? Entscheidung

3.2.1 Monolithischer Kernelaufbau:

Monolithische Kernel sind verhältnismässig ?leicht? zu implementieren: Der Kernel besteht nur aus einem Prozess, im Gegensatz zu einem Mikrokernel ist keine hochperformante IPC und ein hochperformantes Scheduling notwendig, um alleine die Funktionalität und Performance des Kernels sicherzustellen.
Für HalOS ist diese Architektur ausreichend, denn es ist nicht damit zu rechnen dass Hardwarekomponenten oft dazugefügt oder ausgetauscht werden. Des Weiteren liegt der Hauptfokus von HalOS nicht auf Security und Portabilität, was ein weiterer Vorteil von Mikrokerneln wäre. Der Monolithische Kernel muss einfach neu kompiliert werden, wenn er auf einer anderen Plattform betrieben werden soll, oder wenn neue Treiber eingebunden werden sollen.
Zur Laufzeit müssen weder Teile des Kernels aus Speichergründen ausgelagert werden, noch Treiber dynamsich nachgeladen werden, deshalb sind Kernel-Module nicht erforderlich.
Im Source Code werden dann entsprechende Treibermodule und Defines umgeschaltet. Treiber werden direkt in den Kernel kompiliert.
Die Grösse des Kernels sollte überschaubar und wartbar bleiben, deshalb ist die Gefahr den Kernel zunehmend durch Fehlerhafte Treiber zum Absturz zu bringen gering, zumal erwartet wird dass keine Dritthersteller Treiber dafür entwickeln.

3.2.2 Filesystem:

Für persistente Speicherung steht auf dem Board ein SD-Card Slot zur Verfügung. HalOS wird als Filesystem FAT16/32 verwenden, da es dafür schon Implementierungen gibt und der Fokus auf der Entwicklung eines Betriebssystems liegt, nicht auf der Entwicklung eines Filesystems. Im Raw-Format auf die SD-Karte zu schreiben würde sehr starke Einschränkungen mit sich bringen und müsste selbst wieder implementiert werden

(File-Anfänge, Grösse der Files, Berechtigungen? Etc.)
Ausserdem ist es durch die Verwendung von FAT16/32 möglich, von einem Windows-PC mit Card-Reader HalOS-Applikationen und Dateien auf die SD-Karte zu kopieren und in HalOS zu verwenden.

3.3 2) System Calls für Kerneleben:

3.3.1 OS Gedanken zum Startup und was so der Kernel alles machen muss.

1)Startup-File Prozessor (ATMEL) 2)Beim Startup ? wechsel in Kernel Mode 3)Kerenel init (Queue init, 4)Device Driver Framework init (UART, Systemtimer, PIO, LCD, MMC 5)Shell starten

4 Programmier-Richtlinien

So wie Atmel?

http://uplab.hsr.ch/labor/richtlinien/pdf/Coding_Standard.pdf

http://www.jetcafe.org/jim/c-style.html

momo: natürlich gibts noch di gnu coding standardds... was oh ok wären

http://www.gnu.org/prep/standards/

drb8w: wem C zu wenig ist: es gibt auch c+

http://www.state-machine.com/devzone/cplus_3.0_manual.pdf

5 Hardware-Auswahl

Hitachi Displays: (rs-components)

3,5" mit touch

http://at.rs-online.com/web/search/searchBrowseAction.html?method=getProduct&R=6199005

5.7" mit touch

http://at.rs-online.com/web/search/searchBrowseAction.html?method=getProduct&R=6199061

sharp

http://at.farnell.com/1305462/optoelektronik/product.us0?sku=sharp-lq043t1dg01

Question
How do I convert a *.elf file to a *.bin file or to a *.hex file?
Answer
Here is the step by step procedure to follow: * Open a MSDOS command shell * Type: ***avr32-objcopy -O binary myfile.elf myfile.bin*** for a binary file * Type: ***avr32-objcopy -O ihex myfile.elf myfile.hex*** for a hex file

Was bedeuten die Sektions im Hex File:

http://www.roboternetz.de/wissen/index.php/Speicherverbrauch_bestimmen_mit_avr-gcc

6 interprocess communication

Verschiedene Möglichkeiten um eine Kommunikation zwischen Prozessen zu ermöglichen.

1-2 Fragen:

Wie umfangreich muss unsere IPC sein? Genügt ein einfaches Senden von Signalen?? zb. um Prozesse zu killen (SIG 9, u.dgl) Ist ein Kommunikation zwischen User-Prozessen notwendig oder muss nur das System (kernel) mit den Prozessen kommunizieren können? dann: signale wenn user-proc ipc: Sollen wir eine Kommunikation direkt zwishcen User-Prozessen ermöglichen, oder soll jede Kommunikation über den kernel gehen (hat u.u. vorteile)? Benötigen wir eine bidiriktione Kommunikation?

6.1 Pipes

FIFO (in & out) wird für die IPC verwendet.

6.1.1 anonymous pipes

unnamed oder anonymous pipes sind anonyme pipes und bestehen nur für die Lebenszeit eines Prozesses. Die Pipes können nur eine Richtung verwendet werden, dh. wenn in beide Richtungen kommuniziert werden soll, dann werden zwei Pipes benötigt. anonymous pipes werden normal nur bei Prozessverwandschaft verwendet. Dh. ProzessA erzeugt mittels "fork()" einen Sohn-Prozess, Sohn und Vater können nun über eine Pipe (pipe()) kommunizieren.

Unter Linux werden anonymous pipes recht häufig verwendet (mittels "|"), z.b.

```
cat /etc/apache2/conf.d/trac.conf | grep require
```

durch "|" werden die Prozesse "cat" und "grep" von der shell mittels fork generiert.

6.1.2 named pipes

Named Pipes sind Systemweit bekannt, und werden meist in Form eines Files repräsentiert. Named Pipes ist z.b. unter Linux eine sehr wichtige Form der IPC. FIFOs können mit dem Befehl mkfifo() angelegt werden, und werden im Filesystem als "spezielle" Datei abgelegt. Prozesse können auf diese Pipes connecten. Für eine Full-duplex Kommunikation werden wiederum zwei Pipes benötigt.

6.2 Messages

die "mqueue.h" liefert folgede funktionen zum POSIX msg-queueing:

mq_open() -- Connects to, and optionally creates, a named message queue.

mq_close() -- Ends the connection to an open message queue.

mq_unlink() -- Ends the connection to an open message queue and causes the queue to be removed when the last process closes it.

mq_send() -- Places a message in the queue.

mq_receive() -- Receives (removes) the oldest, highest priority message from the queue.

mq_notify() -- Notifies a process or thread that a message is available in the queue.

mq_setattr() -- Set or get message queue attributes.

to be continued.....

6.2.1 Signals

Die Prozesskommunikatoin über Signale ist eine relativ simple Form der IPC und kommt bei UNIX/LINUX zum Einsatz. Signale sind kurze Nachrichten, diese bestehen nur aus einer Nummer und haben keinerlei Argumente (o. dgl.).
Der Kernel stellt hierbei die Funktionalität zur Generierung der Signale zur Verfügung. Es gibt auch einen POSIX standard bzgl. Signale.

6.3 RPC

RPC oder RMI ist ein Möglichkeit der IPC, allerdings denke ich hier eher an ein client-server modell. eine implementierung ist hier eher unwahrscheinlich aufgrund von risiken der aufwandabschätzung.

6.4 Shared Memory

IPC erfolgt über einen gemeinesen reservierten Speicherblock.

6.4.1 Problematik

Gleichzeitges Schreiben: Überschreiben von noch nicht abgearbeiteten Daten; Abhilfe durch Mutual Exclusion (Auer vorlesung??). Softwaremöglichkeiten wären z.b. Locks oder Semaphoren; bekannt sind hier die algorithmen von peterson und dekker (auch dijkstra hat hier was gebaut) Lesen: sind die gelesenen Daten schon gültig? (ereignisbehandlung) busy-wait u.dgl. hier muss ich noch checken wie weit uns der avr unterstützt. Linux arbeitet mit Spinlocks um diese Problematik zu beseitigen (acht geben auf Deadlocks; Versuch von zweimaligem locken von ressourcen).

6.5 Fazit

Das Rad sollte nicht neu erfunden werden. Shared Memory und named pipes sind meiner Meinung eine gute Möglichkeit um eine IPC zu realisieren.

Error: Macro AddComment(None) failed

```
'AddComment' macro can only be used in Wiki pages.
```

6.6 9. Memory Management

6.6.1 9.1.1. Address Binding

Zur **execution time**: Der Prozess kann während seiner Ausführung das Memory-Segment wechseln, das Binding erfolgt zu Laufzeit.

6.6.2 9.1.2. Logical- versus Physical Address Space

MMU mappt logical (vitual) addresses auf physical addresses.

Das user program sieht nur die *logical* addresses, *nie* die *physical* addresses.

6.6.3 9.2. Swapping

Wenn ein high-priority Prozess ankommt, kann der memory manager für einen ganzen anderen low-priority Prozess ein swap out machen. Nach Beendigung des HP Prozesses, kann der LP Prozess mit swap in wieder geholt werden (**roll out, roll in**).

Das OS hat eine ready queue, welche alle Prozesse enthält, deren Speicherimeages im RAM oder auf dem backing store sind. Wenn der Sheduler einen Prozess ausführhern will, callt er den dispatcher. Der dispatcher checkt, oder der Prozess im RAM ist. Wenn nicht, und zuwenig Speicher frei ist, macht der dispachter einen swap out eines anderen Prozesses und einen swap in des gewünschten, lädt die Register und übergibt die Kontrolle.

Nur idle Prozesse dürfen geswapt werden. Wenn I/O ansteht, dann kann der Prozess entweder nicht ausgelagert werden. Alternativ können I/O-Operationen von OS-Buffern bedient werden.

6.6.4 9.4. Paging

6.6.5 9.4.1. Basic Method

Erlaubt, dass der physical-address space eines Prozesses nicht zusammen hängt.

Der physikalische Speicher wird in fixe blocks, sog. **frames**, unterteilt. Der logische Speicher in **pages** derselben Grösse unterteilt.

Jede Addresse der CPU wird in zwei Teile geteilt:

- page number (p)
- page offset (d)

Die page number ist der Index in der **page table**.

Wenn ein Prozess ankommt, muss mindestens seine Grösse in frames verfügbar sein.

Dazu braucht das OS eine **frame table**:

- welche frames sind allokiert
- welche frames sind frei
- wieviele frames insgesammt

Das OS braucht für jeden Prozess eine Kopie der page table.
Diese wird auch vom CPU dispatcher gebraucht.

6.6.6 9.4.2. Hardware Support

Wenn die page table gross ist, liegt sie im Hauptspeicher und ein **page-table base register (PTBR)** zeigt auf sie. Zusätzlich gibt's einen schnellen **translation look-aside buffer (TLB)** mit Einträgen von Key-Value Paaren.

Wenn eine Page-Nummer in der TLB gefunden wird, ist die Frame-Nummer sofort verfügbar. Wenn nicht, gibt's einen **TLB miss** und die page table muss durchsucht werden. Wenn der frame gefunden ist wird er verwendet und zusätzlich in die TLB eingetragen.

Wenn die TLB voll ist muss das OS einen anderen Eintrag rausschmeissen - siehe **10.4. Page Replacement**. TLB Einträge können auch **wired down** sein, d.h. nicht entfernbar.

Address space identifiers (ASIDs) identifizieren einen Prozess. Bei TLBs mit ASID muss die ASID des aktuellen Prozesses mit der der virtuellen page übereinstimmen. Sonst gibt's einen TLB miss. ASIDs ermöglichen Einträge für mehrere Prozesse in einer TLB. Ohne ASIDs muss die TLB bei jedem context-switch geflusht werden.

HALOS: HALOS wirs ASIDs im Rahmen einer Inverted Page Table verwenden. Der TLB des AVR-Systems unterstützt sowohl ASIDs als auch das wired down, welches ev. auch eingesetzt wird.

6.6.7 9.4.3. Protection

Speicherschutz wird mit protection bits in jedem frame erreicht. Normalerweise stehen diese Bits in der page table.

Es gibt:

- read-only
- read-write
- execute-only
- valid/invalid

Valid bedeutet, dass die page im logischen Addressraum des Prozesses ist. Illegale Addressen werden so über das invalid-bit getrappt.

HALOS: HALOS wird die oben angeführten Rechte zum Speicherschutz beachten, da dies vom der TLB des AVR-System unterstützt wird (siehe Access Permissions). Bei einem Fault kann der entsprechende Prozess einfach beendet werden.

6.6.8 9.4.4. Struktur of the Page Table

6.6.9 9.4.4.1. Hierarchical Paging

forward-mapped page table: n-level paging algorithm, wo die page table selber gepaget ist.

Two-level paging: http://kodzilla.org/dok/PRA/Two_Level_Paging%20Levent/

6.6.10 9.4.4.3. Inverted Page Table

Hat Einträge, welche die frames enthält, nicht die pages.

http://www.cs.nmsu.edu/~pfeiffer/classes/573/notes/ipt.html

HALOS: HALOS wird eine Inverted Page Table verwenden, da Memory Sharing zwischen den Anwendungen nicht benötigt wird und es aus Speicherplatzgründen effizienter ist nur eine einzige Page Table mit den wirklich vorhandenen frames zu halten als mehrere Page Tables mit zum grossteil leeren Page-Einträgen. Die Suchgeschwindigkeit in der Inverted Page Table ist auch kein Problem, da die Standardanwendungen sowieso nur beim Ladevorgang des jeweiligen Prozesses auf die Page Table zugreifen werden, nachher wird alles im RAM liegen und über den TLB erreichbar sein.

6.7 10. Virtual Memory

Ist die Separation von logischem zu physikalischen Speicher.

6.7.1 10.2. Demand Paging

Ist wie swapping, nur mit einem **lazy swapper** bzw. **pager**. D.h. niemals eine page swappen, bevor sie gebraucht wird - **pure demand paging**.

6.7.2 10.2.1. Basic Concepts

Der pager schätzt, welche pages verwendet werden.

Zur Unterscheidung welche pages im RAM und welche auf der Disk sind, wird das valid-invalid Schema von 9.4.4. verwendet.

- valid: page ist im RAM
- invalid: page ist not valid oder auf der disk

Zugriff auf eine invalid page löst ein **page-fault trap** ans OS aus.

Schritte:

1. Überprüfe in interner table im process control block (PCB), ob die Referenz valid ist
2. Falls invalid, Termination. Falls valid mache page in
3. Finde freien frame
4. Aktualisiere interne table, dass page im RAM ist
5. Führe unterbrochene instruction erneut aus

Da der Status des unterbrochenen Prozesses gesichert wird, kann er wieder an der selben Stelle gestartet werden.

Harware Support:

- **Page table**: valid-invalid bit
- **Secondary memory**: hält die pages, welche nicht im RAM sind (disk). Siehe Swap-space allocation 14.

Problem, wenn instruction mehrere Speicherstellen modifiziert, da während der Abarbeitung ein page fault auftreten kann, die instruction aber schon Änderungen vorgenommen hat.

Lösungen:

- **Microcode**: Access aller notwendigen Speicherstellen/-grenzen, um notwendige Pages im Vorfeld zu laden
- **Temporäre Register**: bei page fault werden alte Werte wieder zurückgeschrieben

HALOS: HALOS geht davon aus, dass instructions entweder wiederholbar sind, oder dass entsprechender Microcode vorliegt.

6.7.3 10.2.2. Performance of Demand Paging

Um den slowdown aufgrund von paging erträglich zu halten, sollten mind. 2.5 Mio Memory Accesses ohne page fault ablaufen!

Um eine bessere Performance zu erhalten kann beim Prozess Start das ganze file image in den swap space kopiert werden, um demand paging nur von dort auszuführen.

Bei binary files kann demand paging auch direkt aus dem **file system (FS)** gemacht werden. Eine solche page kann bei Bedarf überschreiben und anschliessend wieder vom FS geholt werden.

6.7.4 10.3. Process Creation

6.7.5 10.3.1. Copy on write

Parent und child process teilen zu Anfang die selben pages. Diese sind als **copy-on-write** markiert, d.h. sobald sie

beschrieben werden, wird eine Kopie erzeugt. Nur änderbare pages müssen copy-on-write markiert sein. Über einen **pool** von freien pages werden **zero-fill-on-demand** pages zur Verfügung gestellt.

6.7.6 10.3.2. Memory-Mapped files

Erlaubt einem Teil des **virtuellen** Addressraumes logisch mit einem File verknüpft zu sein. Mapping eines disk blocks zu page(s). Der Initial-Access über demand paging löst einen page fault aus, welcher einen page-grossen Abschnitt aus dem File in eine physikalische page schreibt. Weitere reads und writes werden wie RAM-Access behandelt.

Daten werden nicht notwendigerweise sofort ins file geschrieben.

Erlaubt sharing von Daten, wenn mehrere Prozesse auf dieselbe physikalische page verweisen. Copy-on-write erlaubt eigene Kopien von modifizierten Daten.

6.7.7 10.4. Page Replacement

6.7.8 10.4.1. Basic Scheme

Wenn kein frame frei ist, suchen wir einen, der gerade nicht genutzt wird und geben ihn frei. Wir schreiben den Inhalt in den swap space und aktualisieren die page table.

1. finde gewünschte page auf disk
2. finde freien frame:
    1. verwende freien frame
    2. ohne freien frame, wedende einen page-replacement algorithm um einen victim frame zu wählen
    3. schreibe den victim frame auf disk und aktualisiere page und frame table
3. lies gewünschten frame in den freien frame und aktualisiere page und frame table
4. restart user process

Wenn kein frame frei ist, werden zwei page transfers benötigt.

Um das zu reduzieren wird ein modify bit verwendet. Es wird gesetzt, wenn der Inhalt der page verändert wird. Nur wenn das modify bit gesetzt ist, muss die page auf die disk gesichert werden. Ansonsten kann die page nach dem Löschen von der disk wieder geholt werden.

Probleme des demand pagings:

- frame-allocation algorithm
- page-replacement algorithm

6.7.9 10.4.2 FIFO Page Replacement

FIFO Queue mit allen pages vom RAM. Die page am Kopf der queue wird entfernt, die neue page wird am Ende angefügt.

6.7.10 10.4.3. Optimal Page Replacement

Entferne die page, welche über die längste Zeitdauer nicht verwedet wird.

Schwierig, da Wissen über die Zukunft benötigt wird.

6.7.11 10.4.4. LRU Page Replacement

Assiziiere mit jeder page die Zeit der letzten Verwendung. Wähle die page, welche am länsten unbenutzt ist - **last resently used (LRU)**.

Stack-Implementation: Stack von page Nummern. Wenn eine page referenziert wird, wird sie aus dem Stack entfernt und oben gepusht. So ist die oberste page die letzt-benutzte und die unterste die LRU.

Braucht Hardware-Unterstützung jenseits von TLBs.

6.7.12 10.4.5.2. Second-Chance Algorithm

Ein FIFO Algorithmus. Bei jeder page wird das **reference bit** angeschaut. Bei 0 wird die page entfernt, bei 1 wird das reference bit auf 0 gesetzt, gegebenenfalls die arrival-time zurückgesetz und zur nächsten page gegangen.

Circular-Queue-Implementation: Ein Pointer wandert durch die Queue bis er eine page mit einem 0-reference bit findet. Beim Durchwandern werden alle reference bits gelöscht.

Zu finden unter: http://cse.yeditepe.edu.tr/~sbaydere/courses_new/cse331/files/study2_mt2.html
http://www2.dis.uu.se/~brahim/os1/memory/sld041.htm

6.7.13 10.4.5.3. Enhanced Second-Chance Algorithm

Erweitert durch Betrachtung des reference bit und des modify bits als ein geordnetes Paar.

Klassen:

1. (0,0) weder kürzlich gebraucht noch modifiziert - am besten zu ersetzen
2. (0,1) nicht kürzlich gebraucht aber modifiziert
3. (1,0) kürzlich gebraucht aber nicht modifiziert
4. (1,1) kürzlich gebraucht und modifiziert

HALOS: HALOS wird einen Second Chance Algorithmus versenden, da mehr als genügend RAM vorhanden ist, um Shell und SpaceInvaerds? zu laden. Eventuell wird auf einen Enhanced Second-Chance Algorithmus erweitert. Der TLB des AVR-Systems unterstützt sowohl reference als auch modify bit, was beide Algorithmen ermöglicht.

6.7.14 10.5. Allocation of frames

Unter pure demand paging bekommt der erste Prozess bei Bedarf alle frames und Folgeprozesse eventuell weniger.

6.7.15 10.5.1. Minimum Number of Frames

Bei einem page fault von einer ausgeführten instruction, müssen genug frames verfügbar sein, damit alle pages, welche die instruction referenzieren kann, erreichbar sind. Bei one-level indirect addressing sind das mindestens drei frames, darum müssen wir die Anzahl der indirections begrenzen, z.B. 16, damit maximal 17 frames referenziert werden. Andernfalls entsteht ein trap.

6.7.16 10.5.2. Allocation Algorithms

- **equal allocation**: teile m frames auf n Prozesse mit m/n pr Prozess auf
- **proportional allocation**: Teile verfügbaren Speicher anhand der Grösse der Prozesse auf. Es kann auch eine Aufteilung nach Priorität erfolgen oder nach Grösse und Priorität.

6.7.17 10.5.3. Global Versus Local Allocation

- **global replacement**: erlaubt einem Prozess einen frame aus allen frames, auch denen von anderen Prozessen, zu wählen
- **local replacement**: nur frames von dem eigenen set an frames werden verwendet

6.7.18 10.6. Trashing

Wenn die Anzahl der allokierten frames eines Prozesses unter die minimal notwendige fällt, müssen wir für den Prozess ein **suspend** aufrufen und seine verbliebenen Seiten auslagern. Ansonsten wird der Prozess ständig page-faults erzeugen, das sog. **trashing**.

Um trashing zu vermeiden müssen wir einem Prozess soviel frames er gerade braucht zur Verfügung stellen. Das **locality mode**l sagt, das wenn ein Prozess ausgefürht wird, er sich von einer **locality** zur nächsten bewegt. Eine locality ist eine Menge an pages welche zusammen verwendet werden. Wenn wir weniger frames allokieren, beginnt der Prozess zu trashen.

6.7.19 10.6.2. Working-Set Model

Basiert auf der Lokalitätsannahme. Wir definieren ein **working-set window** *delta*. Wir tracken die *delta* letzt benutzen pages. Wenn eine page länger als *delta* Zeiteinheiten nicht mehr verwendet wird, wird sie entfernt.

Wir errechnen den demand *D* als Summe aller **working-set sizes (WSS_i)**.

Wenn der demand grösser ist als die verfügbaren frames, dann entsteht trashing. Darum muss das OS einen Prozess auswählen und in suspend schicken. Das verhindert das trashing bei maximal möglichem **multiprogramming**.

6.7.20 10.6.3. Page-Fault Frequency

Wir wollen die page-fault rate direkt kontrollieren. Wenn die Rate des Prozesses eine obere Grenze übersteigt, geben wir dem Prozess einen zusätzlichen frame. Wenn die Rate unter die untere Grenze fällt entfernen wir einen frame. Bei ungenügend frames wird der Prozess in suppend geschickt.

HALOS: HALOS wird ein Schema aus proportional allocation, local replacement und Page-Fault Frequency verwenden. Da nur wenig Prozesse vorhanden sind (SpaceInvaders, Shell, ...?), ist es kein performance Problem local replacement zu verwenden. Da alle Prozesse sowieso zur Gänze im RAM Platz finden, ist ein komplizierteres Schema als Page-Fault Frequency nicht nötig.

6.8 14.4. Swap-Space Management

6.8.1 14.4.1. Swap-Space Use

Einfaches swapping hält das ganze **process image**. Paging Systeme speichern nur einzelne pages.

6.8.2 14.4.2. Swap-Space Location

Swap space kann im **file system** oder auf einer separaten **partition** geschehen. Im file system ist es aufgrund der Navigation durch die **directory structure** und **disk-allocation structures** sowie der **external fragmentation** ineffizient.

Eine separate partition, welche von einem **swap-space storage manager** verwaltet wird hat weniger interne Fragmentierung, da die Datenlebensdauern der Daten kleiner als bei files ist.

6.8.3 14.4.3. Swap-Space Management: An Example

Den ganzen Prozess zwischen zusammenhängenden disk regions und memory kopieren.

BSD 4.3: Swap space Preallocationg für alle **text-segments** und **data-segments** des Prozesses garantiert beim Prozess-Start genügend space. Pages werden nur einfal vom file system gelesen und dann nur zwischen swap-space und RAM hin und her kopiert. Zur Optimierung können Prozesse identische text-pages sharen.

HALOS: HALOS wird den gleichen Ansatz wie BSD 4.3 verweden, da dies eine schöne Separation des Prozess-Startes und des weiteren Laufverhaltens ermöglicht.

7 Process Management

Die wichtigen Aufgaben des process Managements im Überblick:

- Prozesserzeugung und Prozessterminierung
- Prozesswechsel
- Verwaltung der Prozesskontrollblöcke
- Prozessablaufplanung und Zuteilung (Scheduling und Dispatching)
- Prozesssynchronisation und Interprozesskommunikation
- Zuteilung von Adressraum an Prozesse (MMU)
- Interrupt- und Trapbehandlung
- Buchführung (Performancecounting, Deadline, ...)

7.1 Prozess Control Block (PCB):

Die Systemkontrolltabelle enthält für jeden einzelnen Prozess ein eigenen PCB. Dieser stellt die wichtigste Datenstruktur eines Prozesses dar. Enthält sämtliche Informationen über einen Prozess die vom OS benötigt werden. Die Inhalte:

- Prozess-ID
- Prozessname
- Prozessstatusinformationen
    - ♦ Programmzähler (PC)
    - ♦ Register
        - ◊ ...
    - ♦ Stackpointer (SP)
    - ♦ Status: (ready, running, blocked [wartet aus IO Geräte oder Usereingabe, ...)
- Prozesskontrollinformationen
    - ♦ Zeiger auf die eigene PageTable
    - ♦ StackButtom
    - ♦ StackSize
    - ♦ Deadline (Echtzeitpriorität): in welcher Zeit muss Prozess vom Scheduler wieder angeworfen werden
    - ♦ Priorität ?
    - ♦ Profilingdaten
        - ◊ Erstellungszeit
    - ♦ UserID (für evtl. späteren Multiusereinsatz)
    - ♦ Dateien: Zeiger auf Struktur
        - ◊ Liste aller geöffneten Dateien
        - ◊ Home Directory
    - ♦ Pointer auf Struktur
    - ♦ IPC: Zeiger auf Struktur zur IPC- und Signalverarbeitung

```
PCB example Code einfügen:
```

```
/*
*********************************************************************************************************
*                                          TASK CONTROL BLOCK
*********************************************************************************************************
*/

typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;           /* Pointer to current top of stack                          */

#if OS_TASK_CREATE_EXT_EN > 0
    void            *OSTCBExtPtr;           /* Pointer to user definable data for TCB extension         */
    OS_STK          *OSTCBStkBottom;        /* Pointer to bottom of stack                               */
    INT32U           OSTCBStkSize;          /* Size of task stack (in number of stack elements)         */
    INT16U           OSTCBOpt;              /* Task options as passed by OSTaskCreateExt()              */
    INT16U           OSTCBId;               /* Task ID (0..65535)                                       */
#endif

    struct os_tcb   *OSTCBNext;             /* Pointer to next     TCB in the TCB list                  */
    struct os_tcb   *OSTCBPrev;             /* Pointer to previous TCB in the TCB list                  */

#if (OS_EVENT_EN) || (OS_FLAG_EN > 0)
    OS_EVENT        *OSTCBEventPtr;         /* Pointer to          event control block                  */
#endif

#if (OS_EVENT_EN) && (OS_EVENT_MULTI_EN > 0)
    OS_EVENT       **OSTCBEventMultiPtr;    /* Pointer to multiple event control blocks                 */
#endif

#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void            *OSTCBMsg;              /* Message received from OSMboxPost() or OSQPost()          */
#endif

#if (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE    *OSTCBFlagNode;         /* Pointer to event flag node                               */
#endif
    OS_FLAGS         OSTCBFlagsRdy;         /* Event flags that made task ready to run                  */
#endif

    INT16U           OSTCBDly;              /* Nbr ticks to delay task or, timeout waiting for event    */
    INT8U            OSTCBStat;             /* Task      status                                         */
    INT8U            OSTCBStatPend;         /* Task PEND status                                         */
    INT8U            OSTCBPrio;             /* Task priority (0 == highest)                             */

    INT8U            OSTCBX;                /* Bit position in group  corresponding to task priority    */
    INT8U            OSTCBY;                /* Index into ready table corresponding to task priority    */
#if OS_LOWEST_PRIO <= 63
    INT8U            OSTCBBitX;             /* Bit mask to access bit position in ready table           */
    INT8U            OSTCBBitY;             /* Bit mask to access bit position in ready group           */
#else
    INT16U           OSTCBBitX;             /* Bit mask to access bit position in ready table           */
    INT16U           OSTCBBitY;             /* Bit mask to access bit position in ready group           */
#endif

#if OS_TASK_DEL_EN > 0
    INT8U            OSTCBDelReq;           /* Indicates whether a task needs to delete itself          */
#endif
```

```
#if OS_TASK_PROFILE_EN > 0
    INT32U           OSTCBCtxSwCtr;         /* Number of time the task was switched in                */
    INT32U           OSTCBCyclesTot;        /* Total number of clock cycles the task has been running */
    INT32U           OSTCBCyclesStart;      /* Snapshot of cycle counter at start of task resumption  */
    OS_STK          *OSTCBStkBase;          /* Pointer to the beginning of the task stack             */
    INT32U           OSTCBStkUsed;          /* Number of bytes used from the stack                    */
#endif

#if OS_TASK_NAME_SIZE > 1
    INT8U            OSTCBTaskName[OS_TASK_NAME_SIZE];
#endif
} OS_TCB;
```

7.2 Zur Verwaltung der PCBs werden verschiedene Datenstrukturen verwendet:

Quelle: ProSeminar Lecutures
http://i30www.ira.uka.de/teaching/coursedocuments/1/3-1_Process-Control-Block.pdf

- **Current Pointer** (Current Pointer zeigt auf den PCB des laufenden Prozess)
- Task Array
  - Das Task Array enthält Pointer auf alle Prozesse des Systems
- PID Hash Table
  - Dient dazu über die PID eine Prozesses auf dessen PCB zuzugreifen
  - Tabelleneintrag beinhaltet Pointer auf je eine PCB
- Main Process Queue
  - Doppelt verkettete zyklische Listen
  - Kopf der Listen ist init_task
  - Enthält alle Prozesse des Systems
  - *next_task, *prev_task
- Run Queue
  - Enthält alle Prozesse deren Zustand
  - Alle Zustand: TASK_RUNNING
- Wait Queues
  - Einfach verkettete zyklische Listen
  - Anzahl bei der Kompilierung nicht bekannt nicht in PCB enthalten
  - Besteht aus Pointer auf PCB und auf nächstes Queue-Element

7.3 Prozessstarten:

Die wichtigsten Punkte bei der Erzeugung eines neuen Prozesses:

- Zuteilung von Speicherplatz
- Initialisierung des Prozesskontrollblocks (PCB)
- Integration in die dynamischen Datenstrukturen zur Verwaltung

- ♦ Zuweisung eines Abschnitts in der Prozesstabelle (PCB in die Prozess-Queue/Ready Queue hinzufügen)
- ♦ Zuweisung einer eindeutigen Prozesskennung (PID)
- ♦ Zuweisung des Zustands "bereit"

Anmerkung zu HALOS: Noch unklar ist die Aufteilung vom Programmimage in RAM und FLASH. Hierzu die Übersicht aus "ARM System Developer?s Guide" S. 496.

7.4 Prozesswechsel:

Ursachen für einen Prozesswechsel:

- Terminierung
- Start eines neuen Prozesses (z.B. Shell startet Spiel)
- Interrupt (z.B. durch Ablauf der zugeteilten Zeit/Scheduling, E/A, Speicherfehler)
- Trap (illigale Operationen, ALU...z.b. bei 0 Division, Befehl, ...)
- Supervisor Aufruf (spezieller Befehl/Syscall/exit)

Aktionen bei einem Prozesswechsel:

1. Abspeichern des CPU-Zustands (Register der CPU, PC ...in den Stack des alten Prozesses speichern / in den RAM)
2. Aktualisierung des Prozesskontrollblocks (PCB, SP anpassen, Prozesszustand ändern... waiting / blocked /...)
3. Verschieben des PCB in eine entsprechende Warteschlange (je nach Prozesszustand)
4. Auswahl eines anderen Prozesses für die Ausführung (Scheduling Algorithmus)
5. Aktualisierung des PCB für den neuen Prozess (Zustand ändern, neue Deadline für Echtzeit Scheduling, ...)
6. Aktualisierung der Speicherverwaltungsstrukturen (Queue?s anpassen, neuer Prozess aus alter Queue herausnehmen und in Running Queue kopieren)
7. Herstellung des CPU-Zustands des neuen Prozesses (Register für CPU hereinladen, PC aktualisieren)

Anmerkung zu HALOS: Bei Punkt 6. (Aktualiserung der Speicherverw.) wird ein Private Virtual Memory Verfahren verwendet. Darum befinden sich in der TLB die ASIDs der einzelnen Prozesse (ASID = PID, sollten dieselbe Bit Anzahl/Länge haben), damit die TLB beim Prozesswechsel nicht komplett erneuert werden muss.

7.4.1

Quelle: http://i30www.ira.uka.de/teaching/coursedocuments/1/3-1_Process-Control-Block.pdf[[BR]]

7.4.2 CPU Programming Modell für Prozesswechsel (PCB Austausch):

Bei jedem Prozesswechsel ist die spezielle Archtektur der AVR32 CPU zu beachten. Wichtig sind dabei die Register der CPU und die diversen Modis (Application, Supervisor, Interrupts [INT 0 bis INT 3], Exception, NMI).

Siehe dazu Atmel Dokument AT32AP7000 bzw. doc32003.pdf auf Seite 24 (6.3 Programming Model, 6.3.1 Register file configuration).

...Link-Register (nur auf den Stack, nicht in die PCB)

8 Ressource Manager

Der Ressource Manager erfüllt als Ressourcenverwalter eine esentielle Rolle des Betriebssystems. Er legt fest, welche Prozesse eine Ressource(Tastatur, Bildschirm, etc) belegen, und auf welche Weise die Ressource belegt ist. Es können atomare Operationen auf den Geräten ausgeführt werden (Datei senden, auf Festplatte schreiben, Tastaturinput), die nicht unterbrochen werden dürfen, und solche, bei denen die Geräte mit den Prozessen durchgewechselt werden können (Teil einer Datei von Festplatte lesen usw.). Der Ressource Manager hat ein zweidimensionales Array, in dem gespeichert wird welche PID eine Ressource auf welche Weise belegt (lock, read, write, execute). Wird eine belegte Ressource angefragt, kann der betreffende Prozess in eine Queue eingefügt werden.

8.1 Deadlock Prevention

Ressourcen müssen freigegeben werden bevor neue Ressourcen allokiert werden.

Werden neue Ressourcen allokiert und es werden 2 Ressourcen oder mehr parallel benötigt, so müssen alle diese Ressourcen parallel allokiert werden.

Bei mehreren "pending Requests" wird dem Prozess der Vortritt gelassen, der am meisten Ressourcen parallel benötigt.

Wenn Deadlocks erkannt werden, werden zwangsläufig Prozesse getötet und ressourcen freigegeben.

Als eine der wenigen "intelligenten" Möglichkeiten, müssen alle Prozesse vor der Ausführung die Ressourcen deklarieren, die sie verwenden werden. Sind vor der Ausführung alle Ressourcenbelegunen bekannt, kann ein Algorithmus ausgetüftelt werden der die Ressourcen in einer Reihenfolge zuteilt, in der keine Deadlocks entstehen.

8.1.1 Deadlock Detection

Ressourcen + Prozesse in einem Ressource-Allocation graph abbilden, dann auf zyklen überprüfen. Zyklus an einer Stelle unterbrechen -> töten

Prozess mit niedrigster priorität wird terminiert, Prozess der am längsten laufen wird? Deadline etc.

Problem wenn immer niedrigster prozess suspendiert wird, anderen ressource zugeteilt wird -> starvation

andere möglichkeiten: bei deadlock Prozess-Rollback: Prozess wird bei Zugriff auf Ressource und daraus resultierendem Deadlock abgebrochen und auf den Status vor dem Zugriff zurückgesetzt und suspendiert.

8.1.1.1

8.1.1.2 Comment by momo on Mon Nov 10 12:23:55 2008

test

Error: Macro AddComment(None) failed

```
'AddComment' macro cannot be included twice.
```

9 Scheduling

Durch Scheduling wird eine Reihenfolge für die Ausführung von Prozessen festgelegt. Die Kriterien für die Reihenfolge können an verschiedenen Strategien orientiert sein. Prozesse werden vom Scheduler in eine oder mehrere **Warteschlangen** eingeordnet.

Der Dispatcher teilt den Prozess im Kopf der aktuellen Warteschlange dem Prozessor zu. Somit ist der Scheduler für die Auswahl des nächst anstehenden Prozesses zuständiger, der Dispatcher ist anschließend für das laden und starten dieses Prozesses zuständig.

9.1 Anforderungen an das Scheduling für HALOS:

_ Einprozessorsystem
_ Prozessmenge dynamisch
_ Ausführungszeiten der Prozesse sind nicht bekannt
_ Verdrängung muss möglich sein
_ Abhängigkeiten zwischen Prozessen muss möglich sein (warten auf I/O Geräte)
_ Prioritäten berücksichtigen
_ Weiche Echtzeit (Deadlines berücksichtigen)

9.2 Geforderte Betriebsarten:

General Purpose Betrieb: Durchsetzung (der Strategie); Gerechtigkeit, Lastausgleich

Echtzeitbetrieb: Dringlichkeit, Termineinhaltung, Vorhersagbarkeit

Diese beiden Betriebsarten **stehen im Widerspruch (Gerechtigkeit und Lastausgleich vs. Echtzeitbetrieb).** Aus diesem Grund ist ein Schedulingverfahren gefordert welches beide Anforderungen abdecken kann, wobei aber jediglich eine weiche Echtzeianforderung besteht.

9.3 Aktivierung des Schedulers:

Zeitgesteuert: Ablauf einer Zeitscheibe
Ereignisgesteuert: ...NOCH NICHT BESPROCHEN (evtl. Abbruch durch externen Interrupt, Syscall-Shutdown-Process,...?)

9.4 Arten des Schedulings:

Langfristiges Scheduling: ...
Mittelfristiges Scheduling: ...
Kurzfristiges Scheduling: ...

9.5 Standard Schedulingstrategien:

FCFS - First Come First Served: Der Scheduler wählt den Prozess aus, der die früheste Ankunftszeit hat und demzufolge schon am längsten wartet.
RR- Round Robin: Das Scheduling ist in "Runden" organisiert. In jeder Runde steht jedem Prozess ein festes Zeitintervall (Zeitscheibe, engl. slot) zur Verfügung. Nach Ablauf der entsprechenden Zeit wird zum nächsten Prozess umgeschaltet. Jeder Prozess erhält die gleiche Zuteilung von Prozessorzeit.
SPN - Shortest Process Next: Der Scheduler wählt den Prozess aus, dessen erwartete Ausführungszeit am kürzesten ist. Dieser Prozess wird nicht von höherrangigen Prozessen unterbrochen.
SRT - Shortest Remaining Time first: Es wird der Prozess mit der kürzesten noch verbleibenden Ausführungszeit ausgewählt. Falls ein Prozess mit einer kürzeren Zeit bereit wird, wird der ausführende Prozess unterbrochen und der Prozessor dem neuen Prozess zugeteilt.
HRRN - Highest Response Ratio Next: Der Scheduler wählt Prozesse aufgrund der normalisierten Durchlaufzeit aus.
Feedback: Es werden mehrere Warteschlangen eingerichtet, in die Prozesse augrund ihrer Ausführungsgeschichte und anderer Kriterien eingeordnet werden.

HALOS Anmerkung: Für HALOS steht der Round Robin Algorithmus in der engeren Auswahl.

> Durch eine variable größe der Timeslices könnten Performance-Auswertungen durchgeführt werden.

wia lauft es ab?...später mehr dazu

Quelle: Stallings, S 527

Weiters ergibt sich durch den RR eine faire Verteilung der Prozessorzeiten. Allerdings ist auch eine weiche Echtzeit gefordert, die möglicherweise mit einem Vorschalten eines minimierten DeadlineFirst? Algorithmus erzielt werden könnte. Timeslices müssen dann mind. doppelt so klein wie "default Abschluss-Sollzeit" der Prozesse mit weicher Echtzeitanforderung sein.

Berücksichtungen beim RR (Verfahren ist abhängig von Zeitscheibenlänge):
Zeitquantum > durchschnittl. Durchlaufzeit ==> Kurze Prozesse werden benachteiligt. Zeitquantum < durchschnittl. Durchlaufzeit ==> Viele Scheduler-Aufrufe

9.6 Echtzeit:

Bisher weitere identifizierte Probleme: Syscalls müssen sehr knapp gehalten werden, siehe auch **HALOS Anmerkung** (vorheriger Absatz).

9.7 Shell

Die shell ist das Interface zwischen Benutzer und System.

Sie wird vom Kernel aus direkt nach dem boot-prozess gestartet.

(der standarduser wird automatisch eingelogged, um das System um Benutzer erweitern zu können?)

9.7.1 Kommandos

start spaceinvaders.exe

* startet einen neuen Prozess aus executable files heraus

showrun

* zeigt alle laufenden Prozesse mit Namen + PID (evtl. cpu last usw)

kill PID

* tötet einen Prozess

ALT+TAB

* dieses Kommando wird auf Treiberebene abgefangen

* es dient zum umschalten zwischen Foreground und Background Prozessen (z.b. zwischen Shell und Spiel) und leitet dadurch auch Tastatureingaben usw. um

http://www.cl.cam.ac.uk/~cwc22/hashtable/

10 DataSheet AP700

Clock Connections für (Timer/Counters, und sonstige Module) Seite 80

11 Nützliche Links

11.1 Übersicht

1. Nützliche Links
    1. OS Design and Implementation
    2. AVR:
    3. Makefiles:
    4. Linker:
    5. Loader:
    6. ELF:
    7. Diverse Sourcen und relevante OS:
    8. Subversion:
    9. Testing and Analysis Tools:
    10. Doxygen:
    11. gratis(?) Ebooks:
    12. Alles fürs Coden:
    13. OO-Entwicklung:

11.2 OS Design and Implementation

http://ivs.cs.uni-magdeburg.de/eos/lehre/WS0708/vl_bs1/

http://www-bs.informatik.tu-cottbus.de/fileadmin/user_upload/lehre/

http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/SysPro/

http://www.cs.binghamton.edu/~nael/cs350/notes/

11.3 AVR:

http://www.avrfreaks.net/

http://www.mikrocontroller.net/

AVR32 32-bit MCU - Application Notes...

11.4 Makefiles:

http://mrbook.org/tutorials/make/

11.5 Linker:

http://www.bilmuh.gyte.edu.tr/gokturk/introcpp/gcc/ld_3.html
http://www.scribd.com/doc/2673366/Using-ld-the-GNU-Linker-RHEL

11.6 Loader:

http://www.tenouk.com/ModuleW.html

11.7 ELF:

http://www.linux-kernel.de/appendix/ap05.pdf[

http://www.x86.org/ftp/manuals/tools/elf.pdf

http://www.sics.se/contiki/ (sourcen für ELF-Loader, siehe elfloader-arch.h, elfloader-avr.c, elfloader.c, ...)

11.8 Diverse Sourcen und relevante OS:

http://www.chris.obyrne.com/yavrtos/ (RTOS für ATmega32, Syscall Beispiele, createTask, taskSwitching, ...)

11.9 Subversion:

http://subclipse.tigris.org/

Eclipse install Link für Subclipse...

keyfile-config für tortoise: http://tortoisesvn.net/ssh_howto

keyfile-config für subclipse: http://www.woodwardweb.com/java/000155.html

11.10 Testing and Analysis Tools:

http://valgrind.org

11.11 Doxygen:

http://download.gna.org/eclox/update

http://www.stack.nl/~dimitri/doxygen/docblocks.html

11.12 gratis(?) Ebooks:

http://knowfree.net/category/it-ebooks/

11.13 Alles fürs Coden:

Inline Assembling, Achtung von AVR nicht AVR32 aber es sind sicher einige wichtig sachen dabei die generell gelten. http://www.nongnu.org/avr-libc/user-manual/inline_asm.html#c_names_in_asm

Linker Files usw. Navigation nicht sehr gut auf dieser Seite aber alles für AVR32 vielleicht was dabei. Da gibts auch ein Link zu RedHat? usw. wo dann wirklich alles zum gnu linker steht http://blog.fosstronics.com/2008/07/04/gnu-linker-concepts-which-every-embedded-systems-programmer-must-know-part-2/

http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/arch/avr32/kernel/sys_avr32.c

Wichtig => Do findandr alles würklich alles! http://www.mikrocontroller.net http://www.avrfreaks.net

FreeRTOS Ap7000 SRC http://ap7x-freertos.wiki.sourceforge.net/

Source check out mit : svn co https://ap7x-freertos.svn .sourceforge.net/svnroot/ap7x -freertos/trunk ap7x-freertos

uCOS-II Da müsst ihr euch registrieren dann könnt ihr das Saugen http://www.micrium.com/

GCC Linker Options: http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Link-Options.html#Link-Options

11.14 OO-Entwicklung:

http://www.state-machine.com/devzone/cplus_3.0_manual.pdf